# Chapter 2

# An Introduction to Software Processes and Project Management

In this chapter, we provide a brief introduction to project management and software processes. These topics go hand in hand, as the software processes that the team follow are planned and managed activities, and also, the choice of software processes influences how the project is managed.

## 2.1 Software Processes

One of the key aspects of software engineering is *modelling*. A *model* is a simplified description of an entity or process. In engineering, models are used to help understand a complex system, and by including only the most relevant details and ignoring the less relevant details, engineers can solve the problem at hand. For example, when modelling an air traffic control system, the size of the wheels and the type of rubber used in the tires of the planes is not relevant, so this information is omitted, even though it is contained in the real system. However, the amount of passengers that a plane can carry may be relevant, as it has an impact on the turnaround time of the aircraft.

A *process* is a set of ordered activities, containing inputs, outputs, activities, and resources, enacted for the purpose of achieving a specified goal. Figure 2.1 depicts a abstract model of a process. A process may contain inputs (e.g. requirements from the customer), outputs (e.g. the software requirements specification, the software itself, user manuals), constraints (e.g. the particular architecture on which the software must run), and resources (e.g. the staff).

Engineering software is itself a process. As such, software engineering processes can be modelled and studied, and much research has gone into studying and comparing different software process models.

**Example 2.1.** Requirements engineering is a process. There are many different recommendations for how to approach the requirements engineering phase of a project, but these all have the same goal: to produce a quality software requirements specification artifact that correctly captures the user requirements.

Similarly, most of the process models for requirements engineering have the following activities:

1. elicitation: determining the requirements of the system, typically via research and communicating with the clients;
2. analysis and modelling: break the problem up into smaller problems that are easier to understand and solve, and model the problem using this break-down;
3. specification: record or document the requirements; and
4. validation: determine whether the specified requirements are correct, complete, and consistent.

Typically, requirements engineering process models specify that following order of these activities:

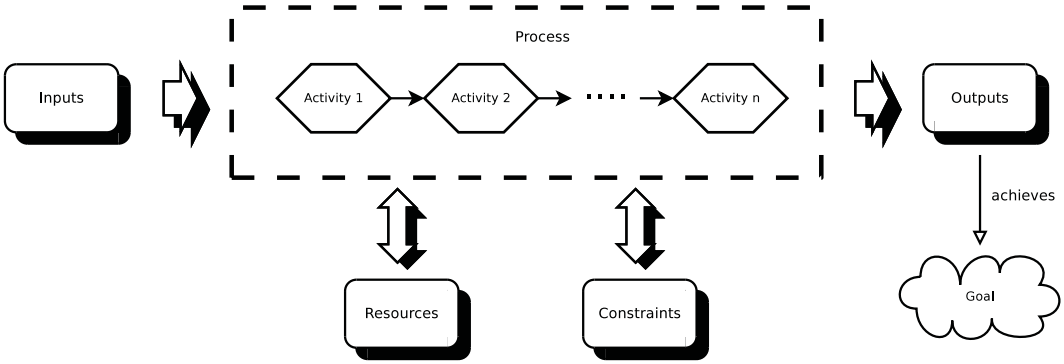
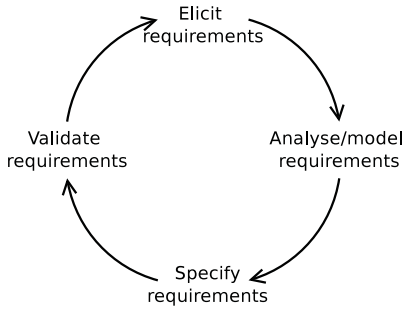
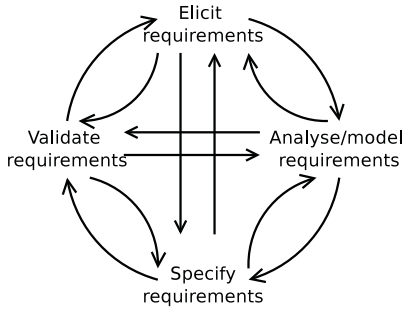



Figure 2.1: An overview of a process. A process is a set of ordered activities that, given some inputs, resources, and constraints, produces some output in order to achieve a given goal.




However, this is only a *model* of the ideal process, and it abstracts away from the details that are not important for understanding how to approach requirements engineering. In reality, the process of requirements engineering in a project looks like this:




That is, at any point in the process, we often return to any other point. For example, it is not uncommon when specifying the requirements to also be doing analysis, or when validating the requirements, to discover a problem requirement that needs to be re-specified. However, as a *model* of the process, the above is almost worthless, because it offers us little insight as to how to achieve our goal.

In a software project, both the goals and the processes that achieve them are hierarchical. Some goals are general, while others are specific. A requirements engineering process model is part of a larger model of the entire project life cycle.

**Example 2.2.** *Software life cycle models*

A *software life cycle model* describes the life cycle of a software project, from conception through to maintenance. It provides a structured blueprint for how a software project should progress. The goal of a life cycle model is to construct and deliver a quality software product.

There is no software life cycle model that fits all projects, but experience and empirical evidence helps us decide which life cycle models are good for certain types of projects.

The *waterfall* model is an example of a software life cycle model. At its most basic, the waterfall model is shown in Figure 2.2. This process model encourages engineers to build the system top-down, progressing onto the new phase of development only after the previous phase is complete. That is, only once the requirements have been complete elicited, specified, and validated, does one progress with design. It gets its name from the fact that the phases cascade like a waterfall.

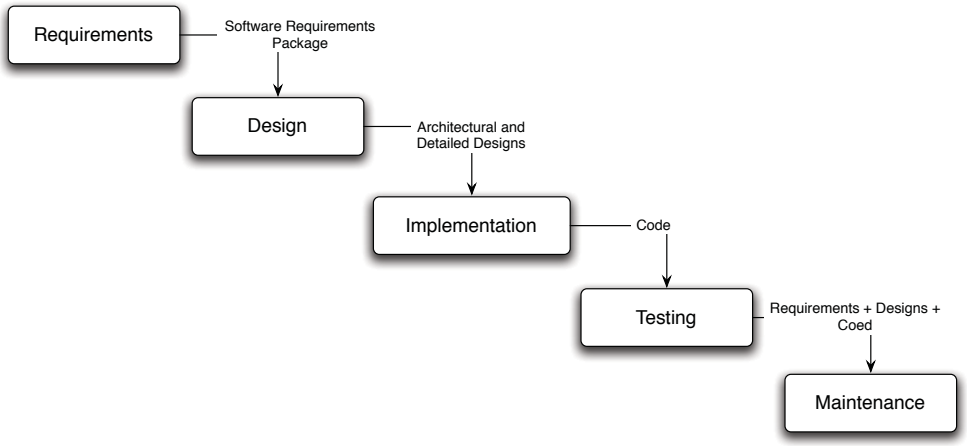



Figure 2.2: The basic model of the waterfall life cycle process. The phases cascade like a waterfall, and each phase only commences once the previous phases is complete.

The waterfall process model is straightforward to understand and apply. However, it has many flaws and many critics. For example, if we follow this process, the system testers do not commence testing until after the implementation is complete. However, this is too restrictive: system testers can start designing test cases and test scenarios when the requirements are being finalised. Starting test case design this early allows testing to be commenced in parallel, saving time. More importantly, however, it allows test engineers to often detect incorrect, incomplete, and inconsistent requirements early in the development process. There is a rule of thumb that says: "If you cannot write a test for a requirement, then that requirement cannot be implemented". Therefore, it is advantageous to start test case design during requirements engineering.

As such, other life cycle models have been proposed, such as the V-model process in Figure 2.3. This process model attempts to overcome the weakness described above in the waterfall model by performing the various levels of testing in parallel with the artifacts against which the testing occurs. For example, the system test case design is done in tandem with the requirements engineering process, so the test engineers can find defects in the requirements before the design is performed, and before the team begin implementation. If a defect is only found during implementation, the design must be re-considered, which is more costly than correcting only the requirement itself.

**Example 2.3.** *Checking source code into a repository*

Goals and process can be low-level, with the process consisting of concrete activities. Checking in some source code into a central repository usually involves a specific process, which will depend on the project and organisation. One such process may be:

1. Update the local working copy of the software from the repository to ensure you have the latest changes.
2. Check that the source-code compiles.
3. Run all unit tests for the part of the system that has changed.
4. Run all systems tests, irrelevant of which part of the system has changed.

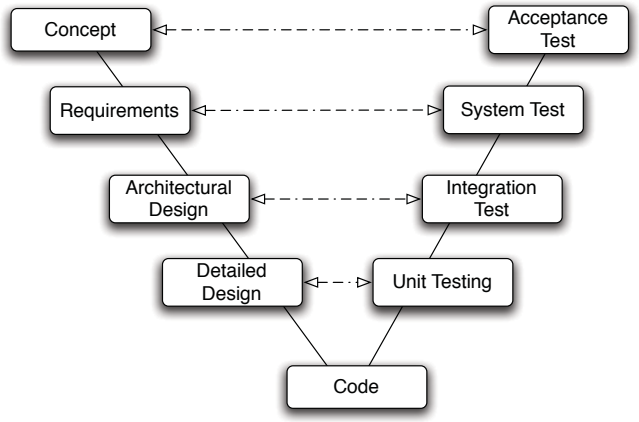



Figure 2.3: The basic V-model model life cycle process. This attempts to overcome some of the weaknesses of the waterfall model from Figure 2.2 by specifying that each level of testing is performed in parallel with the artifacts against which the testing occurs.

5. If *all* of the above steps pass, check in the source code to the repository.

The above achieves the goal of checking source code into the repository and ensures that the updated system will not fail to compile or pass its tests. This way, other developers can checkout a version of the system that compiles and runs.

Each of the example models in this chapter provides a structured way to achieve a goal. These models have been built up over many years using the experience of many different people. When planning a software project, one of the main tasks is to decide *what* the goals of the project are, and *how* we are going to achieve them. With such a large wealth of information available, the job of the managers is to choose the processes that they believe are the best suited to the project and its available resources, and make any changes that they believe are necessary.

## 2.2 Project Management

### 2.2.1 Defining Project Management

The first question we ask is: "What is project management?". The Oxford English Dictionary (OED) defines it as:

> "The theory, practice, or occupation of managing projects."

Err.... yeah, thanks for that one OED. Don't call us... we'll call you.

With such a vacuous definition, we instead ask the OED to define "project" and "management", and define "project management" ourselves:

> "Project: 2c. In business, science, etc.: a collaborative enterprise, frequently involving research or design, that is carefully planned to achieve a particular aim."
>
> "Management: 1a. Organization, supervision, or direction; the application of skill or care in the manipulation, use, treatment, or control (of a thing or person), or in the conduct of something."

Thus, from these two definitions, we define "project management" to be:

> Project management: The application of skill or care in the manipulation, use, treatment, or control of a collaborative enterprise that is carefully planned to achieve a particular aim.

Therefore, project management is about being careful in the way that we run our projects, so that they achieve their aims.

### 2.2.2 Managing Projects via Controlling and Monitoring

In software engineering, projects are managed using two overlapping and related disciplines: *monitoring* and *controlling*.

We monitor the people and resources in the project so we know what is happening and when, so that we can make *timely* and *informed* decisions about the project. We control the project by planning and implementing those decisions.

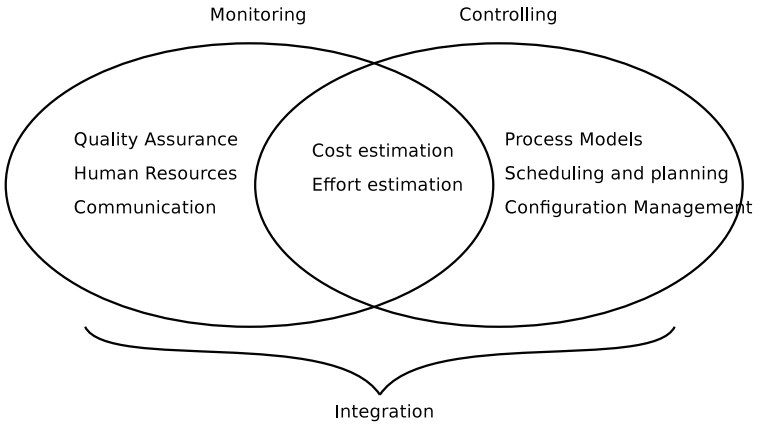
However, these are not distinct phases that the project switches between. We must be constantly monitoring the project to learn what is happening, making decisions based on that monitoring, controlling the project by implementing those decisions, and the monitoring the project to see how the implementation is going. Thus, there is a feedback loop connecting monitoring and controlling. We can define this as:

monitor = decide; control
control = implement decisions; monitor

This simple, formal definition is nice, and it would be great if project management was this easy — well, perhaps not for all those currently employed in project management, who would lose their jobs! Unfortunately, neither monitoring or controlling a project is so straightforward. The mean reason is due to one of the resources that are always involved: human beings.

Humans are a key resource in every project, but we are not very predictable or reliable compared to a physical system. Humans slack off, get sick, quit their jobs, and most of all, make mistakes. All of this means that a plan to control a project will not always go to plan. It further means that monitoring the project is difficult, because the those people doing the monitoring have to rely on other humans to help out, and are also human themselves.

The way that we will look at the disciplines of monitoring and controlling in these notes is to break them into two overlapping sub-groups:




We consider cost estimation and effort estimation as being part of both monitoring and controlling. They are often considered part of the controlling discipline, however, it is inaccurate to say that an estimation of cost or effort controls the project. In fact, it makes as much sense as saying an estimate of how long it takes you to run 100 metres controls how fast you really do run 100 metres. Instead, the cost and effort estimations may *change* the way we control the project, for example, by adding more or less resources; however, once this change has been made, we must redo our estimates based on this new plan. While estimation is clearly not a monitoring activity, we consider it to also be related to monitoring. While cost and effort estimation do not monitor what is happening in the project, they are estimates of what we think our monitoring measurements will be in the future.

The topic of integration, shown in the figure above as collating the two overlapping disciplines of monitoring and controlling, is how to bring all of these activities together into one project.

Throughout this subject, we will learn what each of the above eight activities mean in the context of software engineering.